Porting Code 19

Quickly porting code from other compilers and libraries and from other languages is vital to many people. This chapter describes how some compiler options can help you port code from other compilers and from Symantec C++. It also describes the calling conventions for C and Pascal, in case you're using a library from another language.

Other chapters in this manual also describe how to port code:

- To port assembly code, see Chapter 20, "Writing Inline Assembly."
- To port C++ code, see Chapter 23, "Using Objects in THINK C," especially the section "Differences Between THINK C and C++."
- To port code that uses the ANSI standard libraries, see Chapter 21, "Using the Standard Libraries." If you're porting from a Unix system, look up these topics in the online *Standard Libraries Reference*: "Unix Functions" and "Unix File Functions."
- To port a program without a graphic interface, see the topic "Console Package Intro" in the online *Standard Libraries Reference*.

For more information on how to use the online Standard Libraries Reference*, see "Using the Online Standard Libraries Reference" in Chapter 21.*

Contents

Porting Code to THINK C 369
Changing the size of doubles. 369
Changing the alignment of char arrays in structs 370
Porting code from other versions of THINK C 370

Porting Code to Symantec C++ 370

Calling C Routines 371
C calling sequence 372

C function entry 372
C function exit 373
Functions that return struct, union, or double 373
Functions that accept a variable number of arguments . . 374

Calling Pascal Routines 375
Pascal calling sequence 376
Pascal function entry 376
Pascal function exit 377

Calling C++ Routines 377
Finding the mangled name for a C++ function 378

Porting Code to THINK C

This section describes a few options that make porting code to THINK C easier. The first part describes how to port code from other compilers and the second part describes how to port code from previous versions of THINK C. To port code easily, you use three options in the right column of the Compiler Settings page of the THINK C options dialog, shown in Figure 19-1.

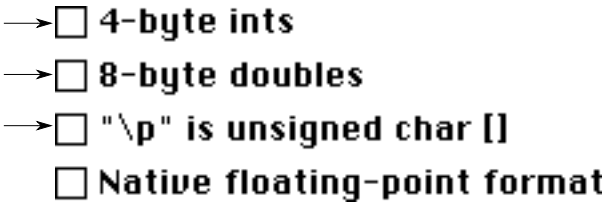

Figure 19-1 Options for porting code

A common problem programmers have when porting code is that the compilers define `int` to be different sizes. By default, THINK C defines `int` to be 2 bytes long, smaller than a pointer. If you're porting code that assumes that `int` is 4 bytes long or the same size as a pointer, you many want to turn on the 4-byte ints option in the Compiler Settings page of the **THINK C options** dialog.

Note

No matter how this option is set, `long int` is always 4 bytes long and `short int` is always 2 bytes long.

Changing the size of doubles

A less common problem is that compilers define `double` to be different sizes. If you're porting code that assumes that `double` is 8 bytes long, turn on the 8-byte doubles option in the Compiler Settings page of the **THINK C options** dialog.

Note

If the 8-byte doubles option is off and the Language Extensions option is off, no type in THINK C gives you 8-byte floating-point numbers.

Changing the alignment of char arrays in structs

By default, THINK C aligns even-length `char` arrays within structures. If you're porting code that assumes that `char` arrays are not aligned within structures, turn off the Align arrays of char option in the Compiler Settings page of the **THINK C options** dialog.

For example:

```
struct S {
   char f1;
   char f2[7];
   char f3;
   char f4[8];
   char f5;
};
```

Normally, THINK C allocates the member `f4` on an even boundary and inserts a pad byte between `f3` and `f4`. The entire `struct` occupies 20 bytes. If the Align arrays of char option is off, THINK C allocates `f4` on an odd boundary and doesn't insert a pad character. The `struct` occupies only 18 bytes.

Note

THINK C always aligns arrays of integers in structures, regardless of how you set this option.

Porting code from other versions of THINK C

Pascal string literals are string literals that start with `\p` *or* `\P`*, for example,* `"\pTitle"`.

Previous versions of THINK C assumed Pascal string literals to be of type `char[]`. If you have code that took advantage of this assumption, turn off the "\p" is unsigned char [] option in the Compiler Settings page of the **THINK C options** dialog.

By default, THINK C now assumes Pascal string literals to be of type `unsigned char []`, which is compatible with the Macintosh Toolbox types `Str255` and `StringPtr`. When you assign a Pascal string literal to a variable of type `Str255` or `StringPtr`, you don't need to cast the string.

Porting Code to Symantec C++

THINK C and Symantec C++ implement most common language features in the same way. In some cases, however, THINK C's default settings are different from Symantec C++'s. If you want to use a C project as a library in a C++ project (such as the ANSI++ library that comes with Symantec C++), use the C++ Compatible button on the

Compiler Settings page of the **THINK C options** dialog. That button turns on two options on the **Compiler Settings** dialog that make your project compatible with Symantec C++. If those options are turned on, the C++ Compatible button is grayed out, as in Figure 19-2.

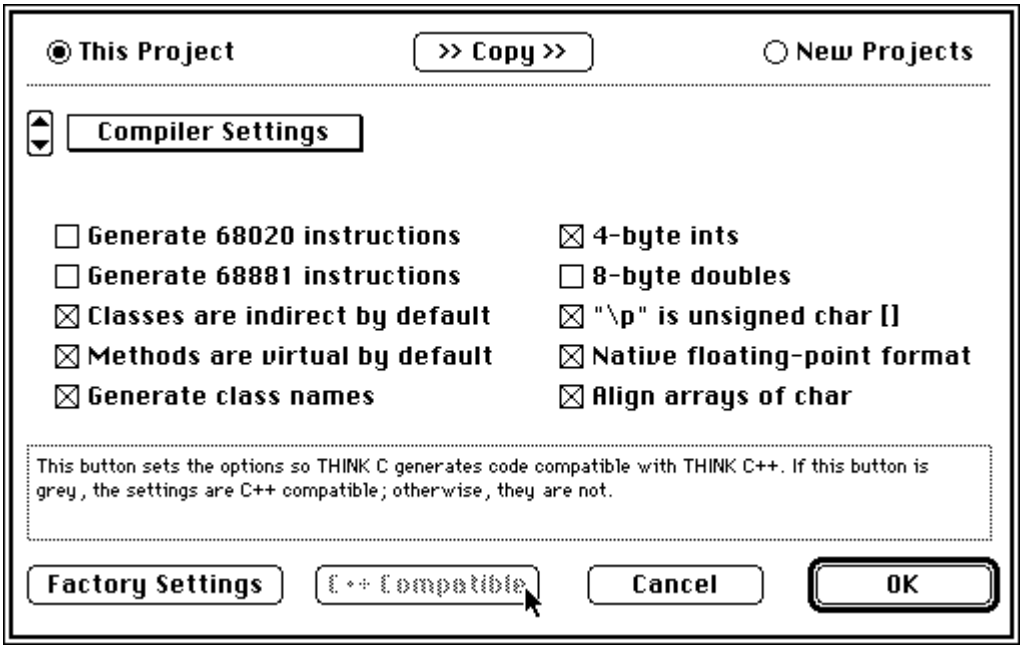

Figure 19-2 The C++ Compatible button

These are the two options that the C++ Compatible button turns on:

- 4-byte ints: Integers declared `int` are always 4 bytes long in Symantec C++. For more information, see “Porting code from other versions of THINK C” earlier in this chapter.
- Native floating-point format: Symantec C++ does not support the universal floating-point format, described in “Floating-Point Representation” in Chapter 15.

Calling C Routines

If you’re using a library written in assembly language or any language other than C, you need to make sure its routines use a

calling convention that THINK C understands. Most of the time you'll use the C calling convention. This section explains how C expects a routine to pass arguments and return values.

THINK C can also understand the Pascal calling convention. The section "Calling Pascal Routines" later in this chapter explains how Pascal expects a routine to pass arguments and return values.

C calling sequence

The caller pushes the arguments in right-to-left order, then calls the function. When the function returns, it's the caller's responsibility to remove the arguments from the stack. The caller's code looks something like this:

```
move  …,-(SP)  ;last argument
…
move  …,-(SP)  ;first argument
jsr   function
add   #…,SP    ;total size of arguments
```

The function's code looks something like this:

```
link  A6,#…    ;(optional)
…
move  …,D0     ;result
unlk  A6       ;(optional)
rts
```

When the Far CODE option is on, the code generator generates 32-bit absolute addresses for the jump table entry of a non-static routine. If the called routine ends up in the same segment as the calling routine, the linker replaces the 32-bit address with a 16-bit relative address from `A5` and a 16-bit `NOP` instruction. This replacement makes it more likely that the linker can remove the jump table entry for that function.

C function entry

The arguments to the function appear on the stack in right-to-left order. The leftmost argument appears just above the return address, followed by the remaining arguments.

The stack looks like Figure 19-3 on entry (just after the call).

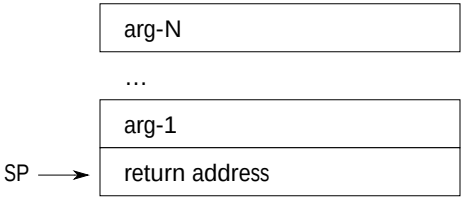

Figure 19-3 Stack during a C function entry

The first argument can be found at `4(SP)`. If the function begins with a `LINK A6,#…` instruction, the first argument can also be referenced by `8(A6)`.

All arguments occupy an even number of bytes on the stack. If there is no prototype for a `char`-sized argument, it is placed in the low byte of an `int`, which may be 2 or 4 bytes, depending on your option settings.

If there is a prototype for a `char`-sized argument, it is placed in the high byte of a `short`.

C function exit

C functions return their result (if any) in register `D0`. The result may be 1, 2, or 4 bytes long. Unused high-order bits may contain garbage.

The stack looks like Figure 19-4 on exit (just after the return).

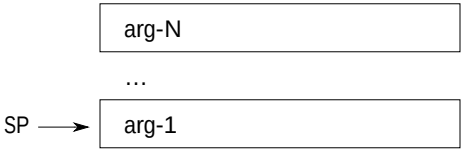

Figure 19-4 Stack during a C function exit

It is the caller's responsibility to remove the arguments from the stack.

Functions that return struct, union, or double

THINK C uses a different method to return a result of type `struct`, `union`, or `double`, since values of these types are in general too large to fit in `D0`. (Some `struct` or `union` types may be small

enough to be returned in `D0`, but the alternate method is used anyway.)

If the Generate 68881 instructions option is on, and the Native floating-point option is on, all floating-point values are returned in register `FP0`.

After pushing the arguments, but before issuing the actual call, the caller pushes the address of the location where the return value is to be placed. This address appears at `4(SP)` (or `8(A6)`) and the first argument appears instead at `8(SP)` (or `12(A6)`). The function must get this address and store the result at the location it points to. The address is considered a hidden argument to the function, and it's the caller's responsibility to remove it from the stack.

Because a function returning a `struct`, `union`, or `double` expects its caller to have placed a hidden argument on the stack, it is essential that the caller do so. Therefore, even when you are not interested in the actual return value, always be sure that the function is declared correctly before calling it.

Note

To ensure that THINK C places the return value in the right place, always use a `return` statement.

Functions that accept a variable number of arguments

The C calling conventions make it easy to write functions that take a variable number of arguments. The first argument can always be found in the same place regardless of how many additional arguments are supplied. Because responsibility for removing the arguments from the stack lies with the caller, the function doesn't need to clean up the stack.

The standard ANSI library includes the `stdarg.h` header file, which provides a standard interface for all C programs that handle variable arguments. As an elementary example, here is a function that returns

the minimum of an arbitrary number of integers. The first argument is the number of additional arguments passed and must be at least `1`.

```
#include <stdarg.h>

int minimum (int count, ...)
{
   va_list xp;              // Declare the
                            //  arg pointer
   int x, min;


   va_start (xp, count);    // Initialize
                            //  the pointer
   min = va_arg (xp, int); // Get the first
                            //  argument

   while (--count) {
      x = va_arg (xp, int);// Get the next
      if (x < min) min = x;//  argument
   }
   va_end (xp);             // Clean up
   return (min);
}
```

Note

You don't need to add the ANSI library to your project to use the `stdarg.h` header file.

A function using Pascal calling conventions cannot accept a variable number of arguments, unless it is written in assembly language.

Calling Pascal Routines

THINK C can use libraries of Pascal routines. When you declare a function with the `pascal` keyword, THINK C expects it to follow the Pascal calling conventions. This section describes how THINK C expects Pascal functions to pass arguments and return values.

This information is also useful if you're writing a callback routine for a Macintosh Toolbox routine.

Pascal calling sequence

The caller pushes the arguments in left-to-right order, then calls the function. Upon return, the result (if any) may be found on the stack. The caller's code looks something like this:

```
CLR   -(A7)        ;reserve space for result
MOVE  #…, -(A7)    ;first argument
...
MOVE  #…, -(A7)    ;last argument
JSR   …            ;go to the function
MOVE  (A7)+, …     ;result
```

The function's code looks something like this:

```
LINK  A6, #…       ;(optional)
...
MOVE  …, …(A6)     ;store return result
. . .
UNLK  A6           ;(optional)
MOVEA (A7)+, A0    ;return address
ADDQ  #…, A7       ;total size of arguments
JMP   (A0)
```

Pascal function entry

The arguments to the function appear on the stack in left-to-right order. The rightmost argument appears just above the return address, followed by the remaining arguments in reverse order. If the function returns a result, space for it is reserved above the first argument. If the return value is 1 byte long, 2 bytes are reserved.

The stack looks like Figure 19-5 on entry (just after the call).

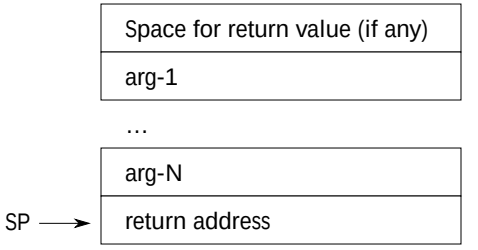

Figure 19-5 Stack during a Pascal function entry

The last argument can be found at `4(SP)`. If the function begins with a `LINK A6,#…` instruction, the last argument can also be referred to as `8(A6)`.

All arguments occupy two or four bytes on the stack. A byte argument appears in the high byte of its word and is found at an even offset from SP (or A6).

Pascal function exit

The function stores its return result (if any) on the stack in the location reserved by the caller. If the result is 1 byte long, it is placed in the high byte of the word reserved.

The stack looks like Figure 19-6 on exit (just after the return).

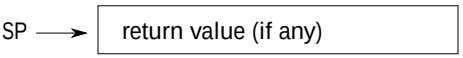

Figure 19-6 Stack during a Pascal function exit

It is the function's responsibility to remove the arguments from the stack.

Note

To ensure that THINK C places the return value in the right place, always use a `return` statement.

Calling C++ Routines

You can call Symantec C++ functions from THINK C code. Since Symantec C++ functions use Pascal calling conventions, call them just like you call Pascal routines, as described in "Calling Pascal Routines" on page 375. The only difference is that you must use the C++ function's mangled name: the name that contains parameter type and return type information. To find a function's mangled name, follow the steps on the following page.

Note

If you declared the function in an `extern "C"` construct in your C++ source file, treat it like any other C function. You don't need to declare it as `pascal` and you don't need to use the mangled name.

For an example of how to use `extern "C"`, see the file `stdio.h`.

Finding the mangled name for a C++ function

If you have Symantec C++, you can find the mangled name for a C++ function by following these steps:

1. Turn on the Generate MacsBug names option in the Debugging page of the **Symantec C++ options** dialog.
2. Open the source file that contains the function you want to call.
3. Choose **Disassemble** from the **Source** menu. A disassembly of the source file appears in a new window.
4. Find your function's definition in the disassembly window. Its mangled name appears towards the end of the function's disassembled code.